ArkDB: A Key-Value Engine for Scalable Cloud Storage Services

Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, Jiesheng Wu
Alibaba Group
Bellevue, WA, USA
{zhu.pang,qingda.lu,s.chen,rui.w,yikang.xu,jiesheng.wu}@alibaba-inc.com

ABSTRACT

Persistent key-value stores play a crucial role in enabling internet-scale services. At Alibaba Cloud, scale-out cloud storage services including Object Storage Service, File Storage Service and Tablestore are built on distributed key-value stores. Key challenges in the design of the underlying key-value engine for these services lie in utilization of disaggregated storage, supporting write and range query-heavy workloads, and balancing of scalability, availability and resource usage. This paper presents ArkDB, a key-value engine designed to address these challenges by combining advantages of both LSM tree and Bw-tree, and leveraging advances in hardware technologies. Built on top of Pangu, an append-only distributed file system, ArkDB's innovations include shrinkable page mapping table, clear separation of system and user states for fast recovery, write amplification reduction, efficient garbage collection and lightweight partition split and merge. Experimental results demonstrate ArkDB's improvements over existing designs. Compared with Bw-tree, ArkDB efficiently stabilizes the mapping table size despite continuous write working set growth. Compared with RocksDB, an LSM tree-based key-value engine, ArkDB increases ingestion throughput by 2.16×, while reducing write amplification by 3.1×. It outperforms RocksDB by 52% and 37% respectively on a write-heavy workload and a range query-intensive workload of the Yahoo! Cloud Serving Benchmark. Experiments running in Tablestore in a cluster environment further demonstrate ArkDB's performance on Pangu and its efficient partition split/merge support.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**; **Transaction logging**; **Database recovery**; **Cloud based storage**.

KEYWORDS

key-value store; cloud storage architecture; transaction logging; recovery; garbage collection

ACM Reference Format:
Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, Jiesheng Wu. 2021. ArkDB: A Key-Value Engine for Scalable Cloud Storage Services. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457553

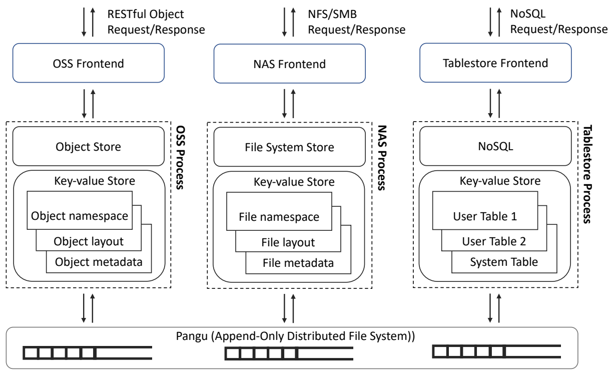

Figure 1: Alibaba Cloud storage service architecture.

1 INTRODUCTION

Nowadays persistent key-value stores [5, 7, 16] play a fundamental role in internet-scale services, which rely on key-value stores to persist system states and facilitate fast lookups. Alibaba Cloud provides a line of fully-managed, scalable and elastic storage services such as Object Storage Service (OSS) [10], File Storage Service (NAS) [9] and Tablestore [11]. These cloud services, available across multiple geographic locations, have become the storage infrastructure of many organizations including Alibaba itself.

Figure 1 shows the overall architecture of Alibaba Cloud storage services consisting of three independently scalable layers: a stateless frontend layer that handles user requests, a service layer that provides service logic backed by a distributed key-value store, and a persistence layer built on an append-only distributed file system named *Pangu* [8]. An efficient and scalable key-value store is vital to service functionality and quality by organizing service states into tables with transaction, backup and snapshot support. For example, to support object and file system operations, Alibaba OSS and NAS maintain object/file namespaces and store the metadata and layout of each object/file in various tables of the underlying key-value store. Alibaba Tablestore manages user data in user-created tables whose access is guided by system tables such as table schemas and access-control lists (ACLs). In this paper, we focus on the design and implementation of the embedded key-value engine in a service process, the central component of the distributed key-value store.

1.1 Requirements and Challenges

The key-value store of Alibaba Cloud storage services faces several important requirements and challenges.

Append-Only Distributed Persistence Layer Along with the evolution of internet information systems, storage disaggregation has become the trend [1, 6, 17, 44], effectively decoupling availability from durability. Since the computation and persistence layers can scale out independently, the resulting system is more cost-effective and better tailored for load balancing. As shown in Figure 1, Alibaba Cloud embraces this approach to decouple the service and persistence layers. We choose to build cloud storage services on a distributed file system that supports append-only writes with truncation [5, 21, 42]. Append-only writes naturally support efficient backups and snapshots. Moreover, fast storage devices, e.g., solid-state drives (SSDs), have been widely deployed in data centers to replace hard disk drives. While modern SSDs exhibit much better writes than hard disks, small random writes are still much slower than large sequential writes and also incur very costly internal fragmentation that shortens SSDs' lifespan [34].

Range Queries Efficient range queries are critical in supporting frequent operations such as listing objects in a bucket or querying log records in a given time frame. Many index structures, while great at point lookups, provide limited range query performance.

Write-Dominant Workloads A large class of user workloads exhibit the behavior of bursts of ingestion and even continuous high-rate ingestion, to the point where many service clusters are dominated by write requests. It is mandatory to provide consistent write performance, which is made challenging by background data compaction and garbage collection on append-only storage.

Scalability A key-value store is partitioned to scale out computation. Multi-tenancy of cloud services results in diverse and dynamic workloads, which demand efficient partition split and merge for load balancing. Besides a partition's writable replica, to scale out reads, multiple read-only replicas need to be maintained efficiently without extra data copies.

Availability At internet scale, system failures are common. After a writable replica fails, a read-only replica is promoted to writable or the writable replica is restarted. Such promotion and restart demand fast recovery for high availability.

Resource Efficiency The key-value engine shares resources with the rest of the service. A resource-hungry index implementation affects service quality and robustness. In a diverse cloud environment, key-value sizes range from tens of bytes to several kilobytes. In case of small sizes, small space overhead per key-value becomes significant with a huge key-value count. For some legacy applications, a single partition may be as large as tens of terabytes, and data accesses are often skewed and most data remain cold after ingested. The resources serving such a large cold partition must be restricted. Thus the engine should adapt to resource constraints.

1.2 Existing Index Technology

Key-value engines support data ingestion, deletion, random point lookup and ordered range query. Two families of index structures provide such capabilities on top of append-only storage.

LSM Tree LSM (log structured merge) tree [36] organizes data into multiple levels, each optimized for its underlying medium. New data changes are buffered in main memory. Once the in-memory data size reaches a threshold, the data are written to the storage sequentially as a new index file, while new data can continue to accumulate in main memory. Both in-memory and on-storage data are indexed to support random point lookups and ordered range scans. To query an LSM tree, the in-memory index and the potentially multiple on-storage indexes are queried with the results merged in the end. To allow sustainable read performance, it is imperative to reduce the number of index files, and hence separate on-storage indexes further into multiple levels. This is accomplished by merging lower-level indexed files into higher-level files in the background. However, such compaction (i.e. merge) activities result in significant write amplification, and often affect user experience.

Bw-tree The other family of designs [4, 28] stems from B-trees with Bw-tree being a notable example. The Bw-tree [28, 29] maintains a logical B+ tree across main memory and append-only storage. It uses log structured store where a page in the Bw-tree does not map to a fixed location on storage. To locate a page, the Bw-tree maintains an in-memory mapping table that maps pages to memory addresses or storage locations, depending on where the most recent version of the page is located. When a page is updated, the new update is captured in a delta that is prepended to the location of the existing page as indicated in the mapping table. When new changes need to be persisted, only the new changes beyond the already-persistent content of the page need to be written to the storage. So a page may consist of a backward chain of updates with newer deltas in memory and older deltas on storage. When a page's update chain contains multiple separately written deltas on storage, the page is deemed fragmented. When page fragmentation reaches a threshold, the page is compacted into a contiguous page. Therefore Bw-tree compaction occurs at page level, in contrast to LSM tree's file level compaction. Page level compaction is mainly triggered on demand upon access. A fragmented page is not compacted when there are no accesses. With changes chained to the corresponding pages, Bw-tree tends to separate hot and cold data. Compaction of a fragmented page does not involve other pages as demanded by file level compaction, incurring less write amplification. Bw-tree's page level compaction puts great demand on random storage reads, which makes it suitable for modern SSDs.

1.3 The ArkDB

An LSM tree-based key-value engine has been the backbone of Alibaba Cloud storage services for over 10 years. However, with exponential data growth, increasing user demands and advances in storage and networking technologies, the existing design has shown its limitations. For example, it does not offer fast range queries due to read amplification from iterating data scattered in different files. High background compaction traffics have made it difficult to maintain consistent write performance. **ArkDB**, a key-value engine was designed to address the above challenges of cloud storage services, with the following innovations.

Two-Tier Architecture ArkDB maintains two tiers. The memory tier buffers newly ingested data. The storage tier maintains an on-storage Bw-tree. A delta in ArkDB represents a page fragment on storage. A query is answered by merging results from the memory and the storage tiers. With this two-tier architecture, ArkDB manages to sustain high-rate ingestion by the memory tier buffering, and reduce write amplification by page level compaction. Bw-tree is also efficient in answering range queries. So ArkDB combines the advantages of LSM tree and Bw-tree.

Shrinkable Page Mapping Table The in-memory mapping table of the Bw-tree is not shrinkable, and thus grows with the B+ tree structure on storage. Each page occupies some memory to track its storage information. Thus the minimal memory occupation by this table is proportional to the tree structure size. In ArkDB, after a page is completely compacted with its index updated in the parent page, the corresponding page mapping entries can be removed. In this way, ArkDB manages to control this table's memory usage.

Two-Level Logging and Recovery In ArkDB, data changes by user transactions are buffered in the memory tier and recorded in the user transaction log. Changes to the page mapping table, including persisting the buffered data changes, page compaction, split and delete, are through system transactions recorded in the system transaction log. This is significantly simplified from a traditional B+ tree based storage engine [23], where the system and user transaction logs are mixed and the page write order must be enforced to control the log size. Restart recovery replays the system transaction log followed by the tail of the user transaction log. Read-only replicas are maintained by continuous replay of the system transaction log (and the user transaction log for more up-to-date reads). Both restart and read-only replica recovery are more efficient than the original Bw-tree design, where the system transaction log and the data are mixed, and the data is scanned for recovery.

Two-Level Snapshot To support snapshot isolation, ArkDB implements multiversion concurrency control without the space overhead of storing a version number with each key-value on storage. A snapshot for reads is defined with both user and coarser system transaction timestamps. To support snapshot queries, page mapping entries are tagged with system transaction timestamps and delta chains are enhanced to *two dimensions*.

Hot-Cold Data Separation ArkDB separates data into hot and cold streams. Such separation reduces write amplification induced by garbage collection (GC). Within each stream, GC does not need to follow the stream order. Instead, the portion in the stream with a higher garbage ratio is garbage collected first. This makes GC more effective. In contrast, the original Bw-tree design has a single data stream and GC simply follows the stream order.

Lightweight Partition Split and Merge ArkDB supports range partitioning with each partition being the unit of work distribution and load balancing. The file system hard-linking capability is utilized to avoid data replication during partition split/merge. Partition split involves cutting the B+ tree pages at the split boundary, and partition merge involves stitching the boundary pages of two trees. Such lightweight cutting and stitching enable efficient partition split/merge that the original Bw-tree design lacks.

In summary, *ArkDB* is an innovative key-value engine architected on an append-only distributed file system. It combines advantages from LSM tree and Bw-tree, and addresses challenges from the cloud serving environment. These are the main *contributions* of this paper. Our experimental evaluation verifies the reduction of memory footprint and storage write amplification, and demonstrates excellent performance on both local and distributed file systems.

The rest of the paper is organized as follows. Section 2 introduces the functional specifications. Section 3 presents the engine architecture. Section 4 describes the core designs. Section 5 describes implementation techniques and optimizations. Section 6 reports the experimental evaluation results with discussion. Section 7 lists the related work. Section 8 concludes this paper. We use **boldface** when defining terms and *italics* for emphasis.

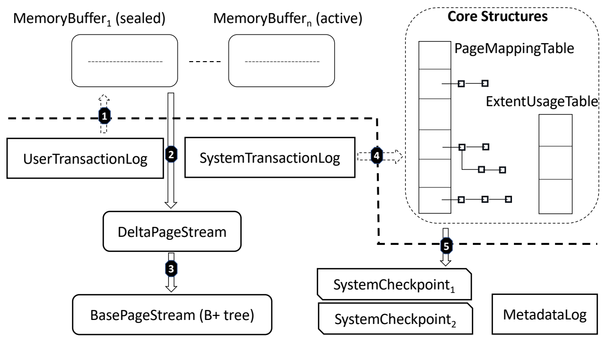

Figure 2: ArkDB two-tier architecture.

2 FUNCTIONAL SPECIFICATIONS

This section describes the function specifications of ArkDB and the underlying distributed file system Pangu.

Data Access Semantics ArkDB supports basic key-value operations including update, point lookup and ordered range scan, where key and value are arbitrary byte strings. With update, a new version overwrites any existing version. A new version with a null value indicates a delete operation. Multiple operations can be wrapped in a single transaction. *Snapshot isolation* [3] is guaranteed such that all reads in a transaction see a consistent snapshot of the store along with the transaction's own data changes.

Append-Only Distributed File System Pangu supports append-only **file streams** with configurable data redundancy from data replication or erasure coding. Each file stream consists of a sequence of append-only **extents**. An extent, identified by a globally unique **ExtentID**, is the erasure unit typically in tens to hundreds of megabytes. Extents within a file stream can be erased individually. Once an extent is sealed, it no longer accepts new data. An extent is the basic unit of sharing among file streams. A file stream can be **hard-linked** to another, with the extents shared. Extents from different file streams can be **concatenated** together to form a new file stream.

3 ENGINE ARCHITECTURE

Figure 2 depicts the architecture of ArkDB, consisting of the upper memory tier and the lower storage tier. A box in the storage tier represents a separate file stream.

Data changes from user transactions are recorded in **UserTransactionLog** and upon commit, applied to the *active* **MemoryBuffer** (❶). A MemoryBuffer supports point lookup and ordered range scan. Once the data volume in the active MemoryBuffer reaches a threshold, e.g., 64 megabytes, this MemoryBuffer is sealed and a new MemoryBuffer is opened to accept new changes. A sealed MemoryBuffer is flushed into the storage (❷). This flushing operation is referred to as taking a **user checkpoint**. The backbone structure on storage is a B+ tree. For a user checkpoint, data changes to the same leaf page form a **delta page** of the **host page**. Delta pages are written into **DeltaPageStream**. A delta page is hooked into the B+ tree indirectly, via a **page mapping** from the host page to this delta page added to **PageMappingTable**. This mapping implies that the actual page content should be obtained by merging the current host page content and this delta page.

When the number of delta pages of a leaf page reaches a threshold, this leaf page and its changes are compacted into a single contiguous page. Later this page's storage location is updated into the parent page, resulting in the parent page being rewritten. Similarly, a page mapping from the old parent page to the rewritten parent page is added into PageMappingTable. This mapping implies the actual content of the parent is now in the rewritten parent page. With the parent page updated, the page mapping entry for the compacted leaf page could be removed. In this way, ArkDB effectively shrinks PageMappingTable. When a host page is compacted, it may be rewritten into multiple pages, in which case, it represents a page split. Both the compacted leaf page and the rewritten parent page are referred to as **base pages** hooked in the B+ tree and written into **BasePageStream** (❸).

Page mapping changes are executed as **system transactions** and recorded in **SystemTransactionLog**. Only after the system transaction log records are persisted, will the changes to PageMappingTable be applied (❹). Conceptually, DeltaPageStream and BasePageStream together with PageMappingTable form an on-storage Bw-tree, a B+ tree variant on append-only storage with delta pages. It is different from the original Bw-tree that is a B+ tree across main memory and storage. A query is answered by merging results from MemoryBuffers and the on-storage Bw-tree.

Along with user checkpoints, page compaction, and page split or merge, extents in DeltaPageStream and BasePageStream are consumed. When a page is rewritten, the prior written parts of this page are invalidated. The exact valid usages of each extent are tracked in **ExtentUsageTable**, which is consulted during garbage collection of DeltaPageStream and BasePageStream. When the memory occupied by ExtentUsageTable reaches a threshold, the historical usage data are compacted, and spilled to storage (BasePageStream indeed) to control the table memory footprint below a threshold.

When the system transaction log size reaches a threshold, both PageMappingTable and ExtentUsageTable are flushed into a **system checkpoint** (❺). After a system checkpoint is taken, its metadata including timestamp and location are recorded into **MetadataLog**. Recovery starts from the most recent system checkpoint indicated by MetadataLog. *Partition split or merge* operations are also recorded in MetadataLog to guide recovery.

4 CORE DESIGNS

The original Bw-tree uses 8-byte *logical* page IDs, which are stored in parent pages. The page mapping table is always searched to find out where the child pages are located in memory or on storage. To our best knowledge, no existing approach can shrink such a page mapping table efficiently to keep its memory footprint small. Moreover, for each page ID, the page's oldest location and some other statistics need to stay in memory, otherwise, garbage collection may incur a significant performance penalty.

With its foundation set on two-tier architecture and two-level logging, ArkDB addresses the core challenge of maintaining a B+ tree with delta pages efficiently on append-only storage in two ways: (1) maintaining a shrinkable page mapping table with concurrent user checkpoint, page compaction and tree structural changes,

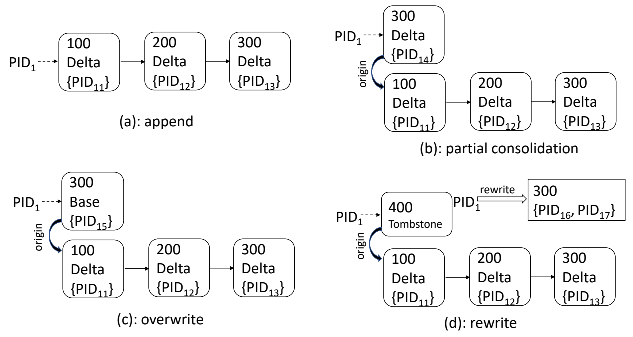

Figure 3: Basic page mapping operations.

and (2) collecting garbage efficiently by tracking valid data with a small memory footprint, and by separating hot-cold data streams to reduce write amplification induced by garbage collection.

4.1 Page Mapping Maintenance

To make PageMappingTable (a concurrent hash table) shrinkable, ArkDB uses *physical* page ID (PID), represented by (ExtentID, Offset, PageSize). For each PID, a *two-dimensional* chain of timestamped page mapping entries is maintained, with each **PageMapping** entry containing the following fields:

- *SysTimestamp*;
- *EntryType*: Base, Delta, Tombstone;
- *Next*: PageMapping*; *first dimension*
- *Origin*: PageMapping*; *second dimension*
- *vector<PID>*;
- *KeyRange. optional, for filtering*

The first dimension is chained in ascending order of *SysTimestamp*. The second dimension enables more granular snapshots. For a page being rewritten, a timestamped entry **PageRewrite** is recorded as (*SysTimestamp*, *vector<PID>*, *vector<HighKey>*) to indicate the resulting pages.

4.1.1 *System Transaction Concurrency.*

Each user checkpoint or batched page compaction is completed in three steps in a system transaction: (step-1) obtaining a timestamp (monotonically increasing counter), (step-2) appending changes to B+ tree into DeltaPageStream or BasePageStream and collecting changes to PageMappingTable, and (step-3) in the timestamp order, writing a log record into SystemTransactionLog to commit this transaction and applying the changes into PageMappingTable. Step-2 is the most intensive step. Multiple system transactions can proceed in parallel at step-2, but are serialized at step-3. Every system transaction makes changes against the snapshot as of a commit timestamp. Given the concurrency at step-2, it is possible that multiple system transactions work against the same snapshot, and make *conflicting* changes, which are reconciled at the step-3. Concurrent changes to the B+ tree structure are coordinated by page level locking.

4.1.2 *Basic Page Mapping Operations.*

Figure 3 exemplifies the basic operations on page mappings along with user checkpoint and page compaction. A user checkpoint flushes data from a MemoryBuffer into DeltaPageStream. When there exist multiple versions of the same key, only the most recent is kept. For each key, it locates the corresponding *host page*. All key-values coming to the same host page are written together as a delta page into DeltaPageStream. The PID of this delta page is **appended** into the page mapping table. A user checkpoint is a heavy system transaction and it is important to allow multiple user checkpoints to run in parallel. In Figure 3-(a), PID_1 appears in this page's parent, and has accumulated three delta pages PID_{11}, PID_{12} and PID_{13}, from three user checkpoints with timestamps 100, 200 and 300, respectively. The content of this page is the merged view from PID_1, PID_{11}, PID_{12} and PID_{13}.

When a page's delta chain length reaches a threshold, e.g., 3, this page is compacted, e.g., in a system transaction with timestamp 400. If the ratio between the total size of delta pages and the base page size has not reached a threshold, e.g., 25%, the delta pages are compacted excluding the base page. This is referred to as **partial consolidation**, which avoids excessive write amplification when this ratio is small. In Figure 3-(b), PID_{11}, PID_{12} and PID_{13} are compacted into a new *delta page* PID_{14}, which has the largest timestamp (300) of the delta pages being merged. The content of this page is the merged view from PID_1 and PID_{14}. On the second dimensional chain via *Origin*, PID_{14} points to those delta pages. When a snapshot query as of 200 comes to PID_1 and finds that PID_{14}'s timestamp is 300, which is larger than 200, the *Origin* of PID_{14} is followed to search further. Thus this second dimensional chain supports a snapshot query as of an old timestamp. The page mapping entries for PID_{11}, PID_{12} and PID_{13} can be deallocated after all snapshot queries no newer than 300 complete.

If the ratio between the total size of delta pages and the base page size reaches a threshold, all the pages are compacted into a new base page to **overwrite** the old base page. In Figure 3-(c), PID_1, PID_{11}, PID_{12} and PID_{13} are compacted into PID_{15} with the timestamp (300) from the largest timestamp of the delta pages being merged. This page's snapshot at 300 is then solely from PID_{15}.

Further, if all the delta pages and the base page are compacted into a number of new base pages AND the number reaches a threshold, e.g., 2, the old base page is **rewritten** into several new base pages. In Figure 3-(d), PID_1 is rewritten into PID_{16} and PID_{17}. PID_{16} and PID_{17} are then posted to the parent page. Meanwhile, PID_1 is marked for deletion by appending a *tombstone* entry with the system transaction's timestamp 400. This tombstone entry's *Origin* points to these delta pages being merged. PID_1's page mapping chain is removed from PageMappingTable after all snapshot queries no newer than 400 complete.

4.1.3 Conflict Reconciliation.

User checkpoints can run concurrently with page compaction against the same snapshot. Conflicting changes are possible and reconciled. In Figure 4, the partial consolidation of PID_{11}, PID_{12} and PID_{13} runs concurrently with the append of PID_{18} against the same snapshot shown in (a). After conflict reconciliation, the append of PID_{18} with timestamp 500 follows the consolidated delta page PID_{15}, as shown in (b).

In Figure 5, the rewrite from compacting PID_1, PID_{11}, PID_{12} and PID_{13} runs concurrently with the append of PID_{18} against the same snapshot shown in (a). The delta page PID_{18} targets the page pointed to by PID_1, which is concurrently rewritten into PID_{16} and PID_{17} with high keys HK_{16} and HK_{17}, respectively. The page mapping table after conflict reconciliation is shown in (b). It is

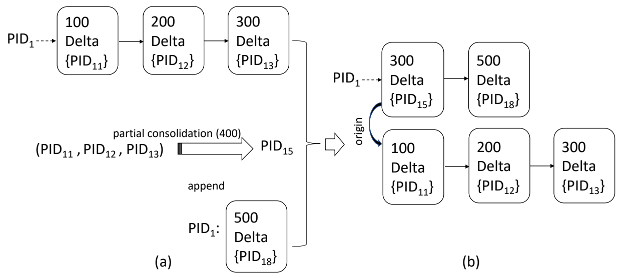

Figure 4: Concurrent append with partial consolidation.

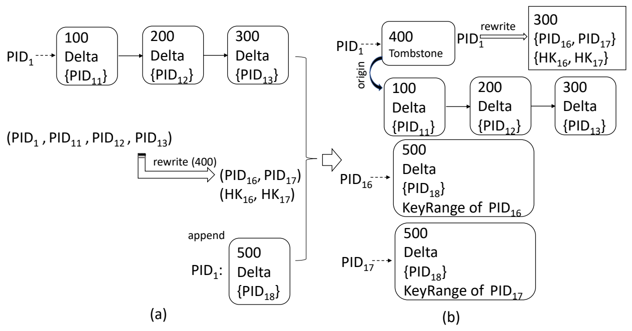

Figure 5: Concurrent append with rewrite.

possible that PID_{18} contains changes in PID_{16}'s key range and PID_{17}'s key range. Therefore, a page mapping entry to PID_{18} is added for both PID_{16} and PID_{17}. Such a page mapping entry uses the key range of PID_{16} or PID_{17} to filter the content of PID_{18}.

The page rewrite information for PID_1 as shown on the upper right corner of (b) needs to be kept around so that if another concurrent user checkpoint working on a snapshot as of 300 tries to append a delta page to the page pointed to by PID_1, this delta page can be correctly *redistributed*. The page rewrite for PID_1 is discarded after all snapshot queries no newer than 400 complete.

4.1.4 Tree Structural Changes.

When a leaf page is compacted, it may be rewritten into multiple *base pages*. When the PIDs of these base pages are posted to the parent (index) page, a *leaf page split* occurs. A leaf page compaction may result in an empty leaf page, in which case, this leaf page's PID will eventually be removed from the parent page. This is a *leaf page delete*.

For search efficiency, when index terms are added into or removed from an index page, this index page is overwritten into one or more *base pages*, similar to Figure 3-(c). If the index page is overwritten into one base page AND this base page's size drops below a threshold, this index page merges with its siblings. This is an *index page delete*. If the index page is overwritten into multiple base pages AND the number of new base pages reaches a threshold, e.g., 2, the PIDs of these base pages are posted further into the parent of this index page. This is an *index page split*. Different from leaf page split shown in Figure 3-(d), index page split is not tracked via page

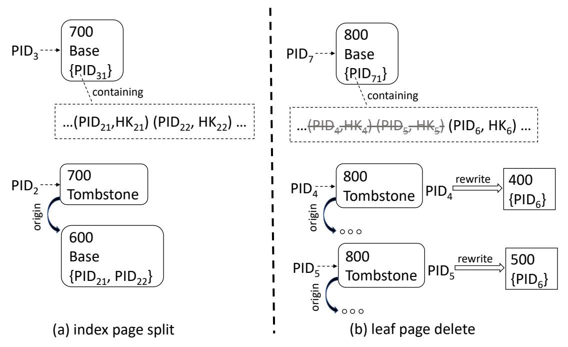

Figure 6: Tree structural changes.

rewrite, since an index page does not have delta pages. After the parent of an index page being split or deleted is updated, the page mapping for this index page is marked for deletion.

Figure 6 exemplifies tree structural changes. In (a), the index page PID_2 is a child of PID_3. PID_2 is updated due to split of its child pages and overwritten into PID_{21} and PID_{22} by a system transaction with timestamp 600. Then PID_{21} and PID_{22} are posted into the parent PID_3 by a subsequent system transaction with timestamp 700. So PID_3 is overwritten into PID_{31}, which contains index terms for PID_{21} and PID_{22}. Meanwhile, the page mapping for PID_2 is marked for deletion by appending a *tombstone* entry.

Deleting a leaf page requires special treatment by adding the rewrite information for the empty leaf page being deleted. In Figure 6-(b), the leaf pages PID_4 and PID_5 are marked for deletion, and they are removed from the parent , which is overwritten into PID_{71}. The next child page in PID_7 is PID_6, which now also covers the key ranges of PID_4 and PID_5. We need to maintain the rewrite information for both leaf pages PID_4 and PID_5 (into PID_6), so that any concurrent append of delta pages to PID_4 or PID_5 can be correctly redistributed to PID_6. One caveat is that in case PID_5 is the rightmost child of PID_7, we write an empty page, called **anchor page**, into BasePageStream. The PID of this anchor page, e.g., PID_{anchor}, is posted into PID_{71}. PID_4 and PID_5 are then rewritten to PID_{anchor}, which covers the key ranges of of PID_4 and PID_5, up to the right boundary of PID_7's key range. PID_{anchor} is discarded when merged with its siblings.

Tree structural changes are in batches and locks are acquired at the page level. Each batch locks an index page and all its child pages being compacted, split or merged. Batching is more efficient than updating the parent page individually for each page split or deletion. With page level locking, multiple tree structural changes can run in parallel.

4.2 Garbage Collection

A key challenge arising from append-only storage is garbage collection (**GC**). The storage usage is committed by system transactions and the usage history is derived from changes to PageMappingTable. In PageMappingTable, when PIDs are placed in the second dimensional chain, i.e., pointed to via the *origin* pointer, the corresponding pages eventually become garbage when no active transactions depend on them. If a system transaction writes some pages into storage, but the system crashes before this transaction commits by persisting the log record for the page mapping table updates, those pages are counted as garbage as well.

ArkDB places delta pages into DeltaPageStream and base pages into BasePageStream. Different from base pages, delta pages are relatively short-lived and tend to be small. This separation of hot data from cold data reduces write amplification induced by GC, since relocation of valid data is reduced. When the garbage ratio of a sealed extent reaches a threshold, this extent becomes a GC candidate. Extents in DeltaPageStream have a higher GC threshold than extents in BasePageStream. Candidate extents are garbage collected in descending order of their garbage ratios.

Similar to user checkpoint and page compaction, GC is completed in system transactions. A GC system transaction checks the snapshot of the underlying B+ tree with delta pages as of a commit timestamp, decides what pages on an extent as of this snapshot are still valid, relocates these pages to the tail of DeltaPageStream or BasePageStream, finally writes a log record to commit the transaction and applies the changes to PageMappingTable.

4.2.1 *Extent Usage Tracking.*

To serve snapshot queries of extent usages for GC, ArkDB tracks the usage history for each extent by maintaining an in-memory data structure **ExtentUsage** with:

- *StartSysTimestamp*;
- *SnapshotSysTimestamp*/*SnapshotUsagePID*;
- *ValidSize*/*ValidRanges*: vector<Range>;
- *UsageLog*: list<IncrementalUsage>.

StartSysTimestamp indicates the timestamp of the system transaction which is the first to commit a write into this extent. *ValidSize* tracks the size of all valid data in the extent as of *SnapshotSysTimestamp*. Each member of *ValidRanges* gives an extent usage snapshot, with each member representing a continuous region of valid data. *UsageLog* is a usage history after SnapshotSysTimestamp and each *IncrementalUsage* records the usage changes on the extent as (*SysTimestamp*, *IncrementalSize*, +vector<Range>, -vector<Range>).

When *UsageLog* is too long, the historical portion no longer needed by GC collapses into a snapshot as of *SnapshotSysTimestamp*. This collapse is completed by a **UsageTrackingCompaction** system transaction and it helps reducing the memory footprint of ExtentUsage. For example, with a valid range snapshot $[10, 19]$ at timestamp 100, followed by a history of $+[20, 29]$ at 110, $+[30, 39]$ at 120 and $-[10, 15]$ at 130. The collapse to timestamp 130 results in a new valid range snapshot $[16, 39]$.

Delta pages tend to be small. The memory usage to track delta pages can be significant. Moreover, delta pages can be redistributed due to page rewrite, thus a delta page can appear in multiple delta chains, as exemplified in Figure 5-(b). This complicates the bookkeeping of the extent usage change history. Thus for extents in DeltaPageStream, ArkDB chooses to track only the valid sizes. Base pages tend to be large and never appear in multiple chains. So for extents in BasePageStream, ArkDB tracks the change history of both valid offsets and sizes. To further limit the memory usage, for an extent in BasePageStream, when the range count after collapse is still beyond a threshold, the UsageTrackingCompaction system transaction spills *ValidRanges* as a **SnapshotUsagePage** into BasePageStream itself, records its location as *SnapshotUsagePID*, and modifies ExtentUsage to reflect the removed usages of the on-storage *ValidRanges* and the added usage of the spilled *ValidRanges*.

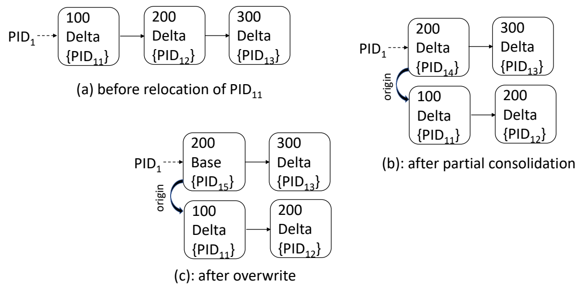

Figure 7: Page mapping changes after page defragmentation.

One caveat is on maintaining the extent usage for delta pages. Due to page rewrite, a delta page can be redistributed into multiple delta chains. ArkDB chooses to count the delta page size once every time when a delta page is redistributed into a delta chain. This delta page size is also deducted once when the delta page is replaced from a delta chain, i.e. being put into the second dimensional chain. In Figure 5-(b), PID_{18} is redistributed to PID_{16} and PID_{17}. It is heavy lifting to find out what percentage of PID_{18} belongs to the key ranges of PID_{16} and PID_{17}, respectively, so the page size of PID_{18} is simply counted twice. A minor side effect is that the reported valid size of an extent in DeltaPageStream can be temporarily larger than the actual size, in rare cases delaying GC of this extent.

4.2.2 Page Relocation. Extent GC relocates valid pages to the data stream tail and skips garbage. ArkDB chooses to consolidate fragmented pages along with GC by potentially relocating the whole chain. The page mapping chains are located differently for base and delta pages. When a base page is written into BasePageStream, a **MappingPID** is embedded in the base page header. For example, in Figure 3-(c), PID_1 is the MappingPID of PID_{15}. For base page PID_b, if there exists a page mapping chain containing PID_b in PageMappingTable, this chain can be located by searching either PID_b or the base page's MappingPID. A delta page may appear in multiple chains, which are located instead by scanning PageMappingTable.

Page defragmentation during GC is a simplified form of page compaction. It does not induce tree structural changes and only the page being defragmented is locked. In contrast, regular page compaction locks the parent pages as well, potentially inducing tree structural changes. The page mapping chains and the parent page paths are implicitly derived for regular page compaction.

Page defragmentation honors the policy of *partial consolidation* to reduce write amplification. In Figure 7, (a) shows the page mapping chain before relocation of PID_{11}. The snapshot read timestamp for this GC system transaction is 200. If the base page PID_1 is located in an extent which is not a GC candidate AND the ratio between the total page size of PID_{11} and PID_{12} and the page size of PID_1 has not reached the threshold, partial consolidation is done with PID_{11} and PID_{12} merged and replaced by a newly written delta page PID_{14} in DeltaPageStream, shown in (b). Otherwise, PID_1, PID_{11} and PID_{12} are merged and replaced by a newly written base page PID_{15} in BasePageStream, shown in (c). PID_1 is locked for this page defragmentation.

4.2.3 GC Flow. ArkDB maintains a global **GCSnapshotTimestamp**, which keeps moving forward with advancing GC rounds. Each round completes in a GC system transaction, which uses the current *GCSnapshotTimestamp* as its snapshot read timestamp and queries the total valid data size of each extent, in order to decide the GC extent candidates. This query can be answered quickly by searching the in-memory ExtentUsages. Any sealed extent with zero valid size can be removed, if its *StartSysTimestamp* is no larger than *GCSnapshotTimestamp*. After a batch of extents are picked as GC candidates, GC finds out the valid ranges of these extents as of the snapshot read timestamp.

For each candidate in BasePageStream, the in-memory ExtentUsage is searched with the result merged with *ValidRanges* located at the *SnapshotUsagePID* in BasePageStream. For all candidates in DeltaPageStream, PageMappingTable is scanned only *once* to construct the valid ranges as of *GCSnapshotTimestamp*. This process is analogous to tracing memory garbage collection used by programming languages [25]. During this process, the page mapping chains containing valid delta pages are also located.

After data in a valid range are loaded into memory, valid pages are identified. For each valid page, the page mapping chains containing this page are relocated with this page (probably partially) defragmented. GC of an extent in BasePageStream may encounter anchor pages and SnapshotUsagePages. An anchor page is handled as a base page. A SnapshotUsagePage for an extent is simply relocated individually with the *SnapshotUsagePID* in ExtentUsage updated in the end. All changes to both PageMappingTable and ExtentUsage are captured in the GC system transaction's log record and applied upon the transaction commit.

5 IMPLEMENTATION AND OPTIMIZATIONS

Based on two-tier architecture and two-level logging, this section further describes implementation and optimization details, including snapshot support, restart recovery, read-only replica recovery, partition split and merge, and finally the caching mechanism.

5.1 Two-Level Snapshot

ArkDB supports the snapshot isolation level. A user transaction reads from a snapshot of the store, merged with the transaction's own changes. ArkDB manages to support snapshot isolation without the need of embedding a version number in each key-value on storage. It eliminates the corresponding space overhead on storage and in memory when the key-values are loaded into the cache. This space saving becomes more obvious for small key-values. To support snapshot read, ArkDB maintains *two levels of correlated timestamps* for user transactions and system transactions.

Each user or system transaction has a separate snapshot read timestamp from its commit timestamp. ArkDB has two timestamp generators for user and system transactions respectively. For a system transaction, its commit timestamp **SysTxTimestamp** is obtained upon transaction start from the system transaction timestamp generator, and its snapshot read timestamp is assigned based on *SysTxTimestamp*. For a user transaction, its commit timestamp **UserTxTimestamp** is obtained upon transaction commit from the user transaction timestamp generator, and its snapshot read timestamp is assigned based on *SysTxTimestamp* and *UserTxTimestamp*.

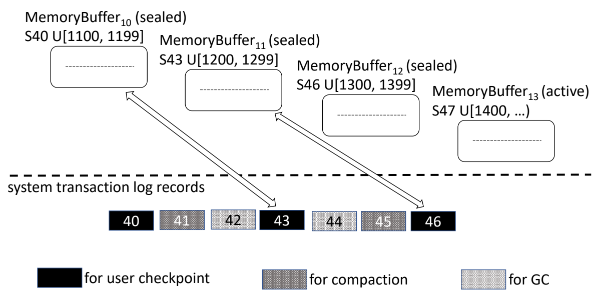

Figure 8: Correlated UserTxTimestamp/SysTxTimestamp.

5.1.1 Timestamp Correlation across Levels. The data changes of user transactions are strictly split among MemoryBuffers by UserTxTimestamps. So each MemoryBuffer contains user transactions in a range of UserTxTimestamps. A new MemoryBuffer is opened when the prior MemoryBuffer stops accepting new data changes AND the user checkpoint system transaction to flush the prior MemoryBuffer to DeltaPageStream has started. This new MemoryBuffer's **MemoryBufferTimestamp**, is set to the *SysTxTimestamp* of the user checkpoint system transaction for the prior MemoryBuffer.

Figure 8 exemplifies the correlation between UserTxTimestamp and SysTxTimestamp. MemoryBuffer$_{10}$ is sealed with data changes of user transactions with timestamps between 1100 and 1199. The prior MemoryBuffer was flushed to DeltaPageStream by a user checkpoint system transaction with timestamp 40. Thus the *MemoryBufferTimestamp* of MemoryBuffer$_{10}$ is 40. Then two system transactions occur with timestamps 41 and 42: one for compaction and the other for GC. MemoryBuffer$_{10}$ is flushed to DeltaPageStream by the system transaction with timestamp 43. Thus MemoryBuffer$_{11}$'s timestamp is 43. MemoryBuffer$_{11}$ is flushed to DeltaPageStream by the user checkpoint system transaction with timestamp 46. MemoryBuffer$_{12}$'s timestamp is 46 and it contains data changes of user transactions between 1300 and 1399. MemoryBuffer$_{12}$ is being flushed to DeltaPageStream, by the system transaction with timestamp 47. This system transaction has not committed yet. MemoryBuffer$_{13}$'s timestamp is 47 and it is accepting new user transactions.

5.1.2 Snapshot Queries. For a SysTxTimestamp TS_s, suppose all system transactions with timestamps no larger than TS_s have committed and among all MemoryBuffers, MemoryBuffer$_x$ has the largest timestamp which is no larger than TS_s, then all new data changes beyond the changes flushed into the storage by TS_s can be found in MemoryBuffer$_x$ and subsequent MemoryBuffers. In Figure 8, beyond the SysTxTimestamp 41, new data changes are located in MemoryBuffer$_{10}$ and subsequent ones; beyond 43, new data changes are located in MemoryBuffer$_{11}$ and subsequent ones.

System transactions are committed in the SysTxTimestamp order, i.e., the commit order in the system transaction log conforms to the SysTxTimestamp order. User transactions can commit out of the UserTxTimestamp order, but are acknowledged to the clients in order. ArkDB maintains **MaxCommittedSysTxTimestamp** and **MinUncommittedUserTxTimestamp**. When a system transaction (except for GC) starts, its snapshot read timestamp is set to *MaxCommittedSysTxTimestamp*. When a user transaction starts, its snapshot read timestamp is set to (*MaxCommittedSysTxTimestamp*, *MinUncommittedUserTxTimestamp* - 1). In Figure 8, if a new user transaction starts, its snapshot read timestamp is set to (46, 1410), assuming the current *MinUncommittedUserTxTimestamp* is 1411. Thus its query is answered with the merged results from the on-storage Bw-tree with PageMappingTable as of 46, and the in-memory MemoryBuffer$_{12}$ and MemoryBuffer$_{13}$ as of 1410. Upon merge, newer changes take precedence over older changes on the same keys. **MinSnapshotReadSysTimestamp** is the minimal snapshot read SysTxTimestamp of all transactions (except GC system transactions). It is used to gate removal of MemoryBuffers and page mapping entries to ensure their availability for snapshot reads. For example in Figure 8, if *MinSnapshotReadSysTimestamp* is 43, only MemoryBuffers up to MemoryBuffer$_{10}$ can be discarded. To ensure availability of extents in DeltaPageStream and BasePageStream for snapshot reads, *GCSnapshotTimestamp* can progress only up to *MinSnapshotReadSysTimestamp*, i.e., 43 here.

Key-values in MemoryBuffers are tagged with UserTxTimestamps to support snapshot queries in MemoryBuffers. To support read-write transactions across multiple partitions, two-phase commit is implemented. To further support snapshot queries across multiple partitions, each cross-partition transaction needs a global timestamp, which can be assigned in a distributed manner [14, 19, 27]. A partition-local transaction does not need a global timestamp, though. In any case key-values on storage do not need to have a version number embedded.

5.2 Two-Level Restart Recovery

To boost restart recovery, user checkpoints and system checkpoints are taken during normal execution. After a MemoryBuffer is sealed, it is flushed into DeltaPageStream by a user checkpoint system transaction. The corresponding system transaction log record contains **UserTxReplayStartInfo**, which includes the current *MinUncommittedUserTxTimestamp* and **UserTxReplayStartPoint**, the minimal position of the first log records of all active user transactions. *UserTxReplayStartPoint* defines the truncation upper boundary of UserTransactionLog after this user checkpoint is taken.

When the system transaction log length reaches a threshold, a system checkpoint is written, comprising: (1) the content of PageMappingTable and ExtentUsageTable as of a *SysTxTimestamp*, referred to as **SysCheckpointTimestamp**, (2) **SysTxReplayStartPoint**, the start position in SystemTransactionLog for system transactions with timestamps larger than SysCheckpointTimestamp, and (3) the *UserTxReplayStartInfo* for the most recent user checkpoint. *SysTxReplayStartPoint* defines the truncation upper boundary of SystemTransactionLog after this system checkpoint is taken. The system checkpoint location is recorded in *MetaDataLog*.

Upon restart, MetaDataLog is read to locate the most recent system checkpoint, from which PageMappingTable and ExtentUsageTable are loaded. Then SystemTransactionLog starting from *SysTxReplayStartPoint* is replayed to apply changes to PageMappingTable and ExtentUsageTable. The B+ tree *root page PID* is included in system checkpoints, and recorded in system transaction log records when changed, thus recovered along with the system transaction log replay. During this replay, newer log records for user checkpoints may be encountered. From the most recent user checkpoint log record, *UserTxReplayStartInfo* (MinUncommittedUserTxTimestamp, UserTxReplayStartPoint) is obtained. Finally, UserTransactionLog starting from UserTxReplayStartPoint is replayed, skipping user transactions with a timestamp smaller than MinUncommittedUserTxTimestamp.

In Figure 8, suppose a system checkpoint is taken as of SysTxTimestamp 44. Replay of SystemTransactionLog starts at the position after the log record for this system transaction 44. The most recent user checkpoint up to 44 is taken by the system transaction 43. However at 46, replay encounters a new user checkpoint system transaction log record, which contains (MinUncommittedUserTxTimestamp=1300, *UserTxReplayStartPoint*). Then replay of UserTransactionLog starts from *UserTxReplayStartPoint*, skipping user transactions with a timestamp smaller than 1300.

A traditional B+ tree based storage engine [23, 35] logs user and system transactions, and the post-images are logged for page split or merge unless a careful page write order is enforced [32], thus more log is scanned for restart recovery than ArkDB, which does not log such post-images. The original Bw-tree design [29] mixes the system transaction log and the data, thus its recovery scans the data after the latest storage checkpoint. ArkDB does not need to scan the data streams for recovery.

5.3 Read-Only Replica Continuous Recovery

To scale out reads, read-only replicas can be maintained on the same copy of file streams, by continuous replay of SystemTransactionLog and if needed, UserTransactionLog.

A read-only replica does not need to maintain ExtentUsageTable. To bring up a read-only replica, *restart recovery* is done with the underlying file streams opened in *read-only* mode. Replaying the tail of UserTransactionLog is skipped, though. Once restart recovery is finished, SystemTransactionLog is continuously replayed to bring PageMappingTable up to date. Queries can read a snapshot as up-to-date as the replay progress. Both the replay progress and the *MinSnapshotReadSysTimestamp* of user transactions on a read-only replica are reported back to the writable replica. On the writable replica, the truncation point of SystemTransactionLog does not advance beyond the replay progress of any read-only replica, and to ensure extents in DeltaPageStream and BasePageStream be available for read on a read-only replica, *GCSnapshotTimestamp* does not advance beyond the reported *MinSnapshotReadSysTimestamp*.

To allow more up-to-date reads, a read-only replica can choose to replay UserTransactionLog simultaneously to apply data changes into MemoryBuffers. To support snapshot queries across MemoryBuffers and the on-storage Bw-tree being restored by replay of SystemTransactionLog, replay of UserTransactionLog should split read-write user transactions into MemoryBuffers in the same way as on the writable replica. So on the writable replica, when a new MemoryBuffer is opened, a **MemoryBufferOpen** log record is written in UserTransactionLog to record its *MemoryBufferTimestamp* and the maximum timestamp of user transactions included in prior MemoryBuffers. With the assistance of *MemoryBufferOpen* log records, replay of UserTransactionLog knows which MemoryBuffers to apply data changes of user transactions.

With concurrent replay of SystemTransactionLog and UserTransactionLog, a read-only replica maintains *MaxCommittedSysTxTimestamp* and *MinUncommittedUserTxTimestamp*, as well. The snapshot read timestamp of a read-only user transaction is set to the current (*MaxCommittedSysTxTimestamp*, *MinUncommittedUserTxTimestamp* - 1). On the writable replica the user transaction log records are generated earlier than the system transaction log record for the user checkpoint which contains the data changes of those user transactions. Thus in most cases UserTransactionLog is replayed ahead of the corresponding part of SystemTransactionLog and a query is answered with more up-to-date data changes from MemoryBuffers. In the rare case that UserTransactionLog replay is behind, a query can be answered without reading from any MemoryBuffer. In Figure 8, if replay of SystemTransactionLog has progressed up to SysTxTimestamp 46, but replay of UserTransactionLog has progressed only to UserTxTimestamp 1180 at MemoryBuffer$_{10}$, a query is answered only based on the replay progress of SystemTransactionLog, effectively as of UserTxTimestamp 1299 (equivalent to SysTxTimestamp 46).

Removal of both MemoryBuffers and page mapping entries on a read-only replica is gated by the *MinSnapshotReadSysTimestamp* of user transactions, a replica-local decision. Replica promotion into writable requires replay to catch up both transaction logs.

5.4 Lightweight Partition Split and Merge

For a range partitioned distributed key-value store, partition split or merge is completed without data replication by utilizing hard links and concatenation capabilities of the distributed file system.

Partition split and merge protocol is similar to what is described in [5]. Splitting a parent partition into two child partitions or merging two parent partitions into one child partition goes through a pre-split or merge stage to prepare the file streams for child partitions. Then each child partition is started by restart recovery enhanced with split or merge of parent B+ tree(s). Partition split or merge is completed after restart recovery of child partitions.

5.4.1 Pre-Split/Merge Preparation. Each partition has a directory on the distributed file system that stores all its file streams. For partition split or merge, there exists a subdirectory to hold the file stream hard links to the parent partitions. Once the preparation for a child partition is started on a parent partition, no new system transactions can be started. After existing system transactions including user checkpoints have completed, a system checkpoint is forced. Then SystemCheckpoints and UserTransactionLogs from the parent partitions are *hard-linked*, and extents in DeltaPageStream and BasePageStream of the parent partition are *concatenated* into the child partition's DeltaPageStream and BasePageStream, respectively. During this process, ArkDB still accepts user transactions, thus new extents can be appended to UserTransactionLogs. Next, a log record is written into MetaDataLog of the newly created child partition to instruct its restart recovery to run with partition split or merge enhancement. Finally, the parent partition is shut down.

5.4.2 Split/Merge Enhanced Restart Recovery. Before restart recovery of a child partition, newly written extents in parent UserTransactionLogs are concatenated into the child's hard links to parent

UserTransactionLogs. Restart recovery first loads SystemCheckpoints from the parent partitions to initialize PageMappingTable (and ExtentUsageTable). In case of partition split, ExtentUsageTable is not maintained during recovery, but reconstructed from scratch.

Next, in case of partition split, for the boundary page at each level of the parent B+ tree, the records falling into the key range of a child partition are extracted and written into a new page, which is linked into the child B+ tree. In case of partition merge, the boundary pages at the same level of the two parent B+ trees are merged, becoming the page at the same level of the child B+ tree. For both partition split and merge, a new root page is generated for each child B+ tree, and a system transaction log record is written to indicate the new root page PID. In case of partition merge, merge of the boundary pages updates ExtentUsageTable. Such update is captured in the same system transaction log record.

The UserTransactionLog tail of each parent partition is then replayed to apply changes into MemoryBuffers of the child partition. In case of partition split, only changes falling into the key range of the child partition are applied. During file stream preparation, if we choose to disallow user writes, the active MemoryBuffer should be flushed before the system checkpoint is taken, and no UserTransactionLog replay is required. In the recovery end, the file stream hard links to parent partitions are removed, and the child partition starts to accept new user transactions.

5.4.3 Post Recovery Processing. After started from split, the child partition conducts a *background* scan on its B+ tree *index pages* as of the recovered largest SysTxTimestamp to: (1) locate and remove page mapping entries not falling into the key range of the child partition, and (2) reconstruct ExtentUsageTable for extents from parent partitions, based on the scan result together with the updated PageMappingTable in (1).

Before the child partition brought up from split/merge is split or merged again, ArkDB ensures all MemoryBuffers generated during recovery have been flushed and then a new system checkpoint has been taken (in case of split, to include the updated PageMappingTable and reconstructed ExtentUsageTable). In this way, a partition to be split or merged has no dependency on its own parents.

5.5 Caching

ArkDB maintains an in-memory page cache for on-storage data. Data pages are of variable sizes and delta pages tend to be short-lived and small. This poses challenges to cache replacement policies. For caching efficiency, ArkDB maintains separate caches for delta and base pages. Both caches are sharded for access concurrency. The base page cache implements the clock page replacement policy [43], while the delta page cache implements the FIFO replacement policy [30, 31]. In each shard the delta page cache uses a fixed-size circular buffer, populated with delta pages in FIFO order. The FIFO policy does not have the effect of the LRU policy that the clock algorithm attempts to emulate, but is efficient in caching short-lived and small delta pages as there is little eviction overhead.

6 PERFORMANCE EVALUATION

ArkDB is implemented in C/C++ as an embedded library running in Alibaba Cloud storage service processes. It is designed to work on Pangu but can also use a local file system for testing and evaluation. This section first describes experimental design and then reports experimental results with discussions.

6.1 Experimental Design

To verify improvements over the original Bw-tree design, we observe the memory footprint of the PageMappingTable, the system transaction log volume relative to data volume, and garbage distribution among extents of DeltaPageStream and BasePageStream. To compare with LSM tree, we measure the performance of ArkDB on a local system using an Ingest-Only workload and two workloads from Yahoo! Cloud Serving Benchmark (YCSB) [13], and compare it against RocksDB (version 6.14) [20], a widely used open-source LSM tree implementation. We integrate ArkDB into Tablestore, a distributed NoSQL storage service on Pangu, then measure the ingestion throughput and verify the benefit of our partition split and merge techniques.

The local experiments are conducted on a dual-socket server with two Intel Xeon Platinum 8276L CPUs @ 2.20GHz, each with 28 physical cores (56 logical cores) and a 38.5MB L3 cache. The system has 192GB DDR4 DRAM and one 2TB Intel P4510 NVMe SSD. The system runs Ubuntu Server with kernel 5.4.0-53. Experiments on Tablestore are conducted in a 9-node cluster, each member system runs a custom Linux image based on kernel 3.10.0-327 and is equipped with dual Intel Xeon Platinum 8163 CPUs@ 2.50GHz with 24 physical cores (48 logical cores) and 32MB L3 cache, a total of 768GB DDR4 DRAM and 10 Samsung PM963 SSDs. The network fabric is 25Gb Ethernet and each system has two bonded network adapters to provide a total bandwidth of 50Gb/s. Pangu is deployed in the same cluster with 3 replicas for all data. Specifications of the SSDs are summarized in Table 1.

Table 1: SSD Specifications

Model:(Rd/Wr)	Sequential(MB/s)	Random(KIOPS)	Latency(μs)
Intel P4510	3200/2000	637/81.5	77/18
Samsung PM963	2000/1200	430/40	85/50

The data set size is 32GB with 200 million records, each with 32-byte key and 128-byte value. The memory cache budget is 8GB, i.e. approximately 25% cache-to-data ratio. For ArkDB this budget is evenly distributed between the delta and base page caches. For RocksDB, this budget is used for block cache including data and index blocks. Each MemoryBuffer or RocksDB memtable is 64MB. Each page or RocksDB block is 64KB. Dependent on the flushing progress, multiple MemoryBuffers or memtables can co-exist. Required by cloud storage services to guarantee data durability, *Direct-IO* is always enabled for local storage and Pangu. RocksDB follows the default configuration unless stated explicitly. Following our engineering practice, ArkDB uses user-level cooperative multitasking for better CPU utilization,while RocksDB's implementation is based on traditional preemptive multitasking supported by OS. For fairness, we configure enough client threads for RocksDB to reach its maximum ingestion performance and in the other cases make sure RocksDB's CPU utilization is always higher than ArkDB's.

6.2 Improvements over Bw-tree Design

Given that no storage-based Bw-tree implementation is available, we compare ArkDB's behavior against Bw-tree's original design.

Figure 9 shows the memory footprint of PageMappingTable along with ingestion of 100 million uniformly distributed key-values on a pre-populated B+ tree with 100 million key-values. The memory usage quickly climbs up and then enters a steady state with fluctuation, around 10MB. The delta chain threshold for page compaction is 4. The count of pages with mapping entries stays around 165K while the total page count reaches 512K. The original Bw-tree would maintain 512K pages in the mapping table. This demonstrates that PageMappingTable size can be effectively controlled. Restricting the memory usage of PageMappingTable is imperative to support large partitions and facilitate fast system checkpoints/recovery.

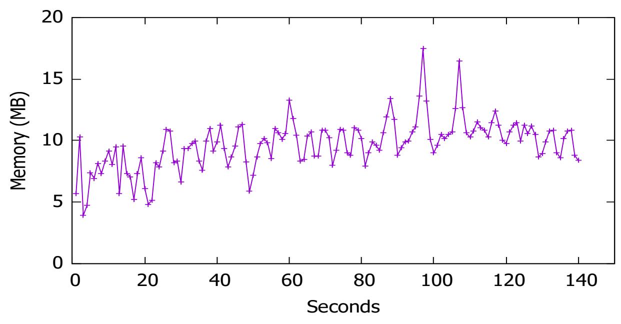

Figure 9: Page mapping memory footprint.

The histogram in Figure 10 shows the distribution of extent garbage ratios in DeltaPageStream and BasePageStream respectively, after ingestion of 100 million key-values without GC. DeltaPageStream contains more extents with a larger garbage ratio. This demonstrates that delta and base pages exhibit different update frequencies and storage separation based on page types reduces write amplification induced by GC. After ingestion, DeltaPageStream and BasePageStream sizes are 27GB and 9.5GB, respectively, while SystemTransactionLog size is only 401MB, about 1% of data volume. Thus recovery of ArkDB is much more efficient than the original Bw-tree which mixes data and the system transaction log.

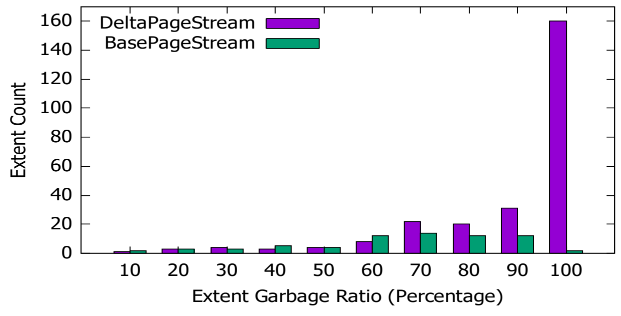

Figure 10: Distribution of extent garbage ratios.

6.3 Comparison with LSM tree

On the server with a local SSD, we measure the throughput of ArkDB and RocksDB with the Ingest-Only workload, YCSB-A (50% updates and 50% point lookups), YCSB-E (5% inserts and 95% scans of average 50 key-values per scan). Each workload starts with a pre-populated B+ tree with 100 million uniformly distributed key-values.

Figure 11 shows the throughput curve for the Ingest-Only workload after 100 million uniformly distributed key-value updates. ArkDB's average throughput is 346 KOPS/sec, about 216% of RocksDB's (average 162 KOPS/sec). The write amplification of ArkDB is 3.1, about 1/3 of RocksDB's (9.5). Because of the significant write amplification reduction from its localized compaction approach, ArkDB also offers better performance consistency: its relative standard deviation is 2.86%, compared to 12.99% with RocksDB. These results demonstrate ArkDB's capability on reducing write amplification and producing consistent write performance.

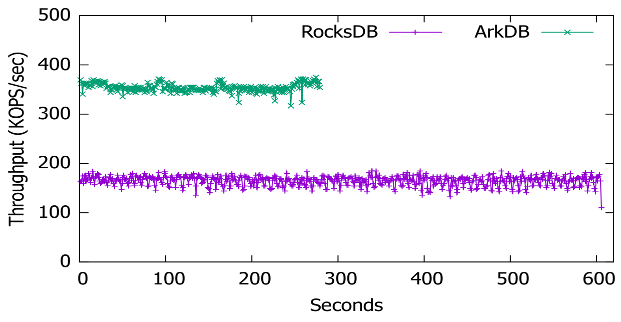

Figure 11: Throughput of Ingest-Only workload.

For YCSB-A and YCSB-E, the keys being seeked are uniformly distributed. Table 2 presents the average throughput after 100 million operations. ArkDB offers higher throughput with both workloads. ArkDB is clearly favored for small range scans as its B+ tree structure and delta chains both preserve spatial locality while LSM tree often has adjacent keys scattered to different files.

Table 2: YCSB throughputs (KOPS/sec)

	RocksDB	ArkDB	Improvement
YCSB-A	269	408	52%
YCSB-E	289	396	37%

6.4 Performance on Pangu

We create a two-column table in Tablestore with multiple partitions, hosted by the same Tablestore server process. Data are ingested into multiple partitions in parallel. Each partition takes 200 million rows of key-values. Higher throughput is achieved by increasing the partition count, as shown in Figure 12 (a). However, the throughput does not scale fully linearly due to contention on the hosting server CPU and on network access to the disaggregated storage. With 16 partitions the total write throughput is 1,349 KOPS/sec, translated to 206MB/sec without considering data replication. In production, a service load balancer would have reacted to such high server loads by splitting or migrating some partitions to less busy nodes.

We next evaluate partition split, a crucial operation to balance loads in production environment. Figure 12 (b) shows the throughput is doubled after a 32GB partition gets split into two 16GB child partitions served by two nodes. During the pre-split stage, incoming requests keep being served, though the writes are to be replayed upon child partition startup. There is a brief period of service unavailability as shown in Figure 12 (b) during the replay stage. In our

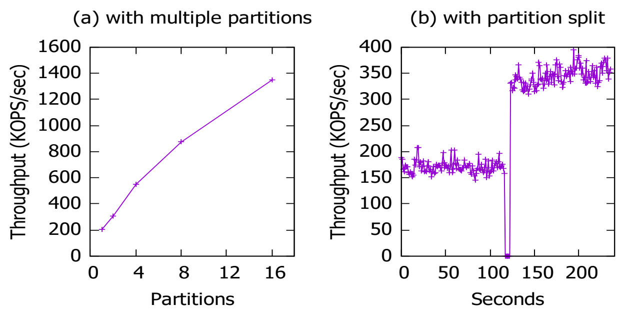

Figure 12: Ingestion throughput of partitions.

test, because the incoming write traffic is very heavy, about 110MB transaction data is built up in the user transaction log and the replay takes a few seconds to complete. The current implementation of ArkDB user transaction replay is single-threaded. We are working on a couple of techniques to minimize the interruption, including speeding up the replay process by parallel replay and shortening the pre-split stage.

7 RELATED WORK

LSM tree. Log structured merge tree was proposed in [36] and made popular by the open-source implementations LevelDB [22] and RocksDB [20]. Recent RocksDB implementation [18] reduces space amplification via compression and write amplification via universal compaction.

Various techniques have been explored to reduce compaction costs. Snow-shoveling compaction was introduced in bLSM tree [40] to increase the amount of data each compaction round consumes, thus reducing the number of compaction runs. In [33], values are stored outside of LSM tree, thus reducing data movement during compaction. Value prefetching is exploited to boost range scans. Inspired by skip lists, Fragmented Log-Structured Merge Trees (FLSM) was proposed in PebblesDB [38]. It organizes sstables at the same level into guards, which have disjoint key ranges and cover the whole key range jointly. Higher levels have more granular guards. Multiple sstables with overlapping key ranges are allowed in a guard, thus reducing data rewriting at the same level. TRIAD [2] reduces write amplification by co-designs among the user transaction log, memtables and sstables. LSM tree's performance is further tuned in a holistic manner by [15] which strikes the optimal balance between update and lookup with the memory budget split by bloom filters and sstable write buffers.

B+ tree on log structured storage. Copy-on-write B+ tree does not modify the original storage, but redirects updates to a leaf page to a new storage area, after merged with the existing data. It is widely used in production systems such as Btrfs [39] and WAFL [24]. Copy-on-write B+ tree fits append-only cloud storage naturally, at the cost of change propagation to the root. Hardware techniques including NVRAM [24] are employed to reduce the frequency of change propagation.

Bw-tree [28, 29] maintains a latch-free B+ tree across memory and log structured storage, and employs a page mapping layer to reduce the cost of change propagation. It is optimized for writes and does page-level compaction, in contrast to the file-level compaction of LSM tree. Storage-backed Bw-tree is used in Microsoft CosmosDB to power its document indexing [41]. Rather than issuing sequential block writes only, [26] sets up a B+ tree directly on an SSD with fixed-size pages mapped to SSD blocks. New changes to a leaf page are accumulated in a corresponding log page, reducing the frequency of updating parent pages.

B$^{\epsilon}$-tree. B$^{\epsilon}$-tree [4] is a B-tree variant with per-page buffers to batch key-values, optimized for writes. New key-values are inserted in the buffer of the root page. After a buffer is filled, key-values are gradually pushed down to the right children until the leaf pages. Bw-tree instead keeps as B+ tree, since only leaf pages can hold values. Tucana [37] uses B$^{\epsilon}$-tree as the basis of its core data structure and integrates several techniques to reduce overheads, such as copy-on-write to avoid write-ahead logging and direct device-memory mapping to reduce space (memory and device) allocation overhead. However, it is nontrivial to extend these techniques to cloud storage.

Composition of data structures. LSM tree consists of memory and storage components that can be different data structures. SplinterDB [12] proposed STB$^{\epsilon}$-tree, combining ideas from LSM tree and B$^{\epsilon}$-tree. Its backbone is a B$^{\epsilon}$-tree. Each trunk consists of a list of branches, each being a B-tree. The root trunk contains a memtable that is an in-memory B+ tree. Compaction occurs at the trunk level. ArkDB bears structural similarities with STB$^{\epsilon}$-tree, as it can be viewed as a two-tier LSM tree with the storage tier in Bw-tree, another B-tree variant. Different from STB$^{\epsilon}$-tree, ArkDB employs page mapping to facilitate fast updates and localize data compaction, and it is architected for append-only cloud storage.

8 CONCLUSIONS

In this paper, we presented ArkDB, a key-value engine architected on Pangu, Alibaba Cloud's distributed file system. Based on a two-tier architecture, ArkDB combines advantages from both LSM tree and Bw-tree. We elaborated on the core designs and algorithms on page mapping maintenance and garbage collection. Furthermore, we described several key implementation techniques: two-level snapshot support, two-level logging, recovery mechanisms including restart recovery and read-only replica continuous recovery, partition split and merge for cluster load balancing, and caching algorithms. Through experiments we verified significant reduction of memory footprint and storage write amplification, and demonstrated excellent performance on both a local system and a Pangu cluster. Alibaba cloud storage services such as Object Storage Service and Tablestore have been updated with ArkDB as the underlying key-value engine. These new implementations are in the production testing phase.

9 ACKNOWLEDGMENTS

ArkDB received continuous support from the Alibaba Cloud Storage Services team. Particularly we appreciate the help and contribution from Wencui Zhou, Jian Huang, Zhongyu Zou, Yufeng Chen, Zhiyong Lin, Shiquan Yang, Ce Yang, Yipu Zhang, Xinjin Chen, Xiaohui Pei, Qi Xu and Weijie Liu. Our thanks also go to our shepherd Dr. Eleni Tzirita Zacharatou.

REFERENCES

[1] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1743–1756. https://doi.org/10.1145/3299869.3314047

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 363–375. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785

[4] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* ACM/SIAM, 546–554. http://dl.acm.org/citation.cfm?id=644108.644201

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 143–157. https://doi.org/10.1145/2043556.2043571

[6] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 205–218. http://www.usenix.org/events/osdi06/tech/chang.html

[8] Alibaba Cloud. 2018. Pangu – The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059.

[9] Alibaba Cloud. 2020. File Storage NAS. https://www.alibabacloud.com/product/nas.

[10] Alibaba Cloud. 2020. Object Storage Service. https://www.alibabacloud.com/product/oss.

[11] Alibaba Cloud. 2020. Tablestore. https://www.alibabacloud.com/product/table-store.

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing.* 143–254.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017.* ACM, 79–94.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220. https://doi.org/10.1145/1294261.1294281

[17] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1463–1478. https://doi.org/10.1145/3318464.3386129

[18] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org.

[19] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013.* IEEE Computer Society, 173–184. https://doi.org/10.1109/SRDS.2013.26

[20] Facebook. 2020. RocksDB. https://rocksdb.org/.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, Michael L. Scott and Larry L. Peterson (Eds.). ACM, 29–43. https://doi.org/10.1145/945445.945450

[22] Google. 2020. LevelDB. https://dbdb.io/db/leveldb.

[23] Goetz Graefe. 2012. A survey of B-tree logging and recovery techniques. *ACM Trans. Database Syst.* 37, 1 (2012), 1:1–1:35. https://doi.org/10.1145/2109196.2109197

[24] Dave Hitz, James Lau, and Michael A. Malcolm. 1994. File System Design for an NFS File Server Appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings.* USENIX Association, 235–246. https://www.usenix.org/conference/usenix-winter-1994-technical-conference/file-system-design-nfs-file-server-appliance

[25] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

[26] Bo-Kyeong Kim and Dong-Ho Lee. 2015. LSB-Tree: a log-structured B-Tree index structure for NAND flash SSDs. *Design Automation for Embedded Systems* 19 (March 2015), 77–100.

[27] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8878)*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer, 17–32. https://doi.org/10.1007/978-3-319-14472-6_2

[28] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 302–313.

[29] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB* 6, 10 (2013), 877–888.

[30] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf

[31] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[32] David B. Lomet and Mark R. Tuttle. 2003. A Theory of Redo Recovery. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 397–406. https://doi.org/10.1145/872757.872806

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* USENIX Association, 133–148.

[34] Changwoo Mina, Kangnyeon Kimb, Hyunjin Choc, Sang-Won Leed, and Young Ik Eome. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 139–154.

[35] C. Mohan and Frank E. Levine. 1992. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, Michael Stonebraker (Ed.). ACM Press, 371–380. https://doi.org/10.1145/130283.130338

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[37] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 537–550. https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

[38] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 497–514.

[39] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3 (2013), 9:1–9:32. https://doi.org/10.1145/2501620.2501623

[40] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 217–228.

[41] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin J. Levandoski, and David B. Lomet. 2015. Schema-Agnostic Indexing with Azure DocumentDB. *PVLDB* 8, 12 (2015), 1668–1679.

[42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[43] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern Operating Systems (4th Edition)*. Pearson Education, Inc.

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1041–1052. https://doi.org/10.1145/3035918.3056101